# Toteutusdokumentti

Ohjelma koostuu yhdeksästä luokasta: App on sen käynnistävä luokka, jossa luodaan muut luokat ja annetaan alkupisteen ja loppupisteen arvot, sekä valitaan mitä kolmesta algoritmista halutaan sillä kertaa ajaa. Piste-luokka sisältää koordinaatistopisteet, joita algoritmi käyttää pohjanaan.

AstarAlgoritmi-luokka sisältää itse haku-algoritmin, joka toimii kuten yleisesti tunnettu A*. Se tuntee myös luokan AstarVertailija, jota se käyttää yhdessä keon kanssa antamaan kulloinkin lähimmän pisteen maaliin nähden.

Djikstran algoritmi on hyvin samanlainen kuin astar, mutta hieman hitaampi. Toisaalta se löytää varmemmin optimaalisen reitin, sillä se ottaa huomioon myös kuljetun matkan. Usein käytetään kuitenkin astari sen ollessa niin paljon nopeampi eikä tuloskaan ole kaukana optimaalisesta.

BFS on leveyssuuntainen haku kahden pisteen valilla. Se on hyvin hidas verrattuna Djikstraan ja Astariin.

Etäisyys on laskettu linnuntietä, mutta pisteiden välillä voi liikkua vain joko ylös, alas, vasemmalle tai oikealle. Tämä ei ole optimaalinen tapa rakentaa heuristiikka, mutta tein sen harjoituksen vuoksi ja nähdäkseni miten se käyttäytyy tälläisessa tilanteessa. Optimaalisesti etäisyys laskettaissin samalla tavalla kuin miten liikkuminen pisteiden välilläkin on toteutettu.

Keko on itse toteutettu tietorakenne, joka jäljittelee javan PriorityQuen toimintaa. Se tarvitsee toimiakseen sisälleen listan, joka myös on tässä itse toteutettu. Lista on tehty toimimaan samalla tavalla kuin javan ArrayList toimisi vastaavassa tilanteessa. Keolle annetaan vertailija, jonka mukaan se järjestää sisältämänsä alkiot. Tässä työssä alkiot ovat pisteitä, joiden etäisyys maalipisteeseen lasketaan. Tämä on toteutettu kahdella eri tavalla. Astarilla linnuntietä ja djikstralle siihen on lisätty jo kuljettu matka.

Huonoimman tapauksen aikavaativuus astarille on $O(|V| + |E|) \log |V|$ ) ja huonoimman tapauksen tilavaatimus on $O(|V|)$. Djikstralla aikavaatimukset ovat huonoimmassa tapauksessa samat kuin astarilla. Keskimääräinen haku-aika on kuitenkin ratkaiseva. Tässä E on polkujen määrä ja V solmujen määrä. Vastaavat BFS:lle ovat $O(E + V)$ ja tilaa $O(V)$ eli solmujen määrän verran.

Työtä vois parannella esimerkiksi muuttamalla pisteiden välistä liikkumista monimutkaisemmaksi. Esimerkiksi sallimalla liikkeet myös väli-ilmansuuntiin ja luomalla esteitä tai/ja hidasteita koordinaatistoon. Myös tietorakenteita voisi parannella niin, että ne toimisivat useammissa tilanteissa. Eli lisätä metodeja, joita on esim javan ArrayListilläkin.

Ohjelmaa on lähdetty tekemään hyvin suoraviivaisesti. Ensin loin piste-luokan vertailijan, jotta voin navigoida jonkinlaisessa koordinaatistossa suunnittelemallani Astar-haulla. Tämän jälkeen itse algoritmin kirjoitus oli jokseenkin helppoa. Seuraavaksi aloin toteuttaa omia tietorakenteita javan valmiiden tilalle. Syntyi lista ja keko. Tässä vaiheessa huomioni kiinnittyi testaukseen, jota olin ajanpuutteen takia lykännyt. Saatuani testauksen tyydyttävään kuntoon siirryin itse dokumenttien kirjoittamiseen loppuhiontaan. Työ ei ollut kuitenkaan vielä valmis, vaan lisäsin siihen vertailun vuoksi vielä kaksi haku-algoritmia: Djikstran ja leveyssuuntaisen haun, jotta kävisi selkeästi ilmi miksi Astar on tällä hetkellä useimmiten paras vaihtoehto polunetsintään.

Lähteet:


http://bigocheatsheet.com/
http://en.wikipedia.org/wiki/A*_search_algorithm
http://en.wikipedia.org/wiki/Breadth-first_search
http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
http://www.cs.helsinki.fi/node/81607
Tietorakenteet ja algoritmit materiaal - Patrik Florell
www.github.com – aikaisemmat työt.